SPECIFICATION

TITLE OF INVENTION

Virtual Supercomputer

CROSS-REFERENCE TO RELATED APPLICATIONS

This application incorporates by reference, in whole, the provisional application:

Virtual Supercomputer
Application No. 60/461,535 Filing Date April 9, 2003
Inventors Gary. C. Berkowitz & Charles C. Wurtz

This application incorporates by reference, in whole, as a partial embodiment of some of the elements of the virtual supercomputer, the prior filed co-pending application:

Knowledge-based e-catalog procurement system and method
Application No. 10/215,109 Filing Date August 8, 2002
Inventors G. C. Berkowitz, D. Serebrennikov, B. M. Roe, C. C. Wurtz

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT

Not Applicable

REFERENCE TO SEQUENCE LISTING, A TABLE, OR A COMPUTER PROGRAM LISTING COMPACT DISK APPENDIX

Not Applicable

BACKGROUND OF THE INVENTION

Technical Field of Endeavor

The present invention is generally directed to the field of computing and more specifically to the field of information processing, and even more specifically, to the field of high-demand, high-performance, and supercomputing.

Description of the Related Art

The ability to use computers to process extremely complex or large amounts of stored information and derive new useful information from that information previously stored has assumed an important role in many organizations. The computer-aided methods used to store and derive information vary, but the vast majority of such methods depend on a general purpose computing hardware architecture. Accordingly, the potential to craft an optimal solution to a computing problem is limited by a static hardware architecture that is optimized not for any specific computing task, but rather to provide an acceptable level of processing ability over a wide and disparate range of computing tasks.

Attempts to create optimal solutions to solve specific and complex information processing tasks have focused on creating hardware architectures designed to exploit various features of the information to be processed so that such processing can be performed in an optimal manner. Hardware devices containing specialized vector processing units are one such example. Software written for such hardware formats the information into a form that takes advantage of the hardware's specialization, thus creating a computing environment that is tailored to the specific problem at hand. Such tailored solutions are usually implemented on high-end supercomputing hardware architectures with specialized software. Consequently, this approach is prohibitively expensive for most organizations, often costing millions of dollars. Additionally, once created, tailored solutions of this type are only suitable for a specific problem or class of problems.

The software written to solve specific high-performance computing problems is necessarily constrained by the features of the hardware upon which such software runs. When such software is in machine-readable form, it is tightly coupled to a specific

architecture upon which it will run. Further, the underlying hardware machine architecture is almost always static, and only reconfigurable (and only partially so) in a few non-commercial machines not widely available. Even the so-called grid (or network or large-cluster) computing approaches, which rely on large numbers of interconnected physical or virtual machines, are still constrained by running on a few different types of conventional processors. While the topology of the network can be configurable in such approaches, the architecture of the underlying processors is static, and thus not tailored to the problem at hand.

The concept of a virtual supercomputer addresses these shortcomings. The virtual supercomputer provides a conceptual, reconfigurable hardware architecture for high-performance machine-readable software. The conceptual hardware architecture masks the actual underlying hardware from the machine-readable software, and exposes to the software a virtual machine reconfigurable for the problem at hand. The virtual supercomputer thus provides to the software the operations needed for optimal processing, unconstrained by the overhead associated with those hardware operations of the underlying machine that are not relevant to the task. This not only speeds the computing required for the problem, but also importantly, it dramatically speeds the process of software application development, as the developer can write program code to a machine that directly processes operations specifically optimized for the problem to be solved.

The virtual supercomputer translates the software instructions from the format for the virtual machine into a format that a particular underlying hardware architecture can process. Each specific hardware architecture must have a specific virtual machine associated with it. Thus, software for the virtual supercomputer can run on a wide variety of hardware

architectures, because the virtual machine for each specific hardware architecture provides the same conceptual hardware architecture for software developers. Therefore, a large investment in a supercomputer or supercomputing cluster, with attendant maintenance and obsolescence issues, is avoided. Further, unlike a grid or conventional network computing system, which increases power in a brute-force manner by simply adding more processors, each virtual machine in a virtual supercomputer network has an internally configurable architecture, thus magnifying the power of the virtual supercomputer to provide a tailored solution.

One embodiment of some portions of the virtual supercomputer is described in the pending application Knowledge-based e-catalog procurement system and method, listed in the CROSS-REFERENCE section of this application.

BRIEF SUMMARY OF THE INVENTION

The present invention solves the previously mentioned disadvantages as well as others. In accordance with the teachings of the present invention, a computer-implemented method, apparatus and system is provided for crafting high-performance information processing solutions that are able to be tailored to specific problems or classes of problems in a way that such tailored solutions will perform on a variety of hardware architectures while retaining the benefits of a tailored solution that is designed to exploit the specific information processing features and demands of the problem at hand.

The present invention provides a reconfigurable virtual machine environment upon which a tailored solution to a specific problem (including a class of problems) is crafted. Additionally, an operating system for such a virtual machine is included. The information to be processed for a problem is encoded into a solution-space, or manifold of nodes, where a

node can be any kind of data structure, and the nodes may be independent (point clouds), or connected in any kind of topology, such as an acyclic directed graph (ACG) structure, a balanced tree, or other suitable data representation. This data representation is specifically constructed to closely match the architecture of the problem to be solved and the information to be processed. By exploring the data representation, the information comprising the problem is processed, and various possible solutions to the problem are generated and evaluated. The identified solution is not necessarily the optimal solution to the problem, but is sufficiently accurate and robust to be useful. The exploration of the data representation is performed in a controlled manner to locate a solution.

In one embodiment of the present invention, the virtual supercomputer operates on a single hardware processor and provides a software environment in which tailored solutions to multiple problems and/or problem classes can be created. In another embodiment of the present invention, the virtual supercomputer operates on a distributed interconnected network of hardware processors. Such processors may or may not be all of the same type. In this second embodiment, the advantages of additional computing resources and concurrent processing can be exploited to find a solution in a highly efficient manner.

BRIEF DESCRIPTION OF THE DRAWINGS

Throughout, a synonym sometimes used for the present invention (virtual supercomputer) is the acronym NVSI (Netcentric Virtual Supercomputing Infrastructure).

FIG. 1 is a block diagram depicting the overall architecture of an embodiment of the virtual supercomputer system.

FIG. 2 is a block diagram depicting the virtual machine's major components and their interconnection.

FIG. 3 is a pseudocode representation of the Configuration Engine.

FIG. 4 is a pseudocode representation of the Instantiation Engine.

FIG. 5 is a pseudocode representation of the Population Engine.

FIG. 6 is a pseudocode representation of the Navigation Engine.

FIG. 7 is a pseudocode representation of the Evolution Engine.

DETAILED DESCRIPTION OF THE INVENTION

The detailed description of the present invention incorporates, in whole, the following two attached documents: the NVSI Virtual Machine Technical Manual, ver 3.0 rev 03/2002 (24 pages), and the technical white paper NetCentric Virtual Supercomputing, ver 1.21 rev 12/13/01 (28 pages).

The accompanying drawings, which are incorporated in and form part of the specification, illustrate an embodiment of the present invention and, together with the detailed description, serve to explain the principles of the invention.

In a preferred embodiment of the present invention, shown in FIG. 1, the virtual supercomputer is a system, apparatus and method, composed of the NVSI Virtual Machine (VM), which is the actual reconfigurable virtual hardware processor, an associated operating system (NVSI-OS), a virtual-machine assembler (NVCL Assembler), an application programming interface (NVSI-API), Platform Drivers, and a Platform Assembler.

A problem domain-specific application requests specific processing tasks be performed for it by the virtual operating system running on the NVSI virtual machine (VM). These processing requests take the form of function calls that are defined by the virtual

supercomputer's application programming interface (API). The architecture does allow for an embodiment in which direct calls to the VM are made by the Domain Application.

The virtual operating system (NVSI-OS) is composed of multiple layers containing a plurality of sub-components. The uppermost layer contains the OS managers. The managers coordinate various aspects of the creation of the solution space and the operation of the virtual supercomputer. Managers manage various engines and can invoke the operation of any set of engines to accomplish a task. The next layer contains engines, daemons, and a toolbox. The engines implement low-level machine instructions to send to the virtual machine and generate code that will activate the virtual machine. Daemons are background processes responsible for such tasks as reconfiguring the data representation, garbage collection, and memory recapture. An example would be pruning of unused or outdated branches in a tree manifold by the navigation engine (see below). The toolbox is a collection of routines that are frequently called by the managers. To accomplish certain frequently preformed tasks, a manager has the option of issuing an instruction to an engine or instead making a call to the toolbox.

The solution space is the collection of nodes or other data formats that are interconnected in such a way as to construct a data representation, or manifold, with input data encoded into its topology. One possible embodiment for such a data representation is an acyclic directed graph. Other possible embodiments include, but are not limited to: independent point-clouds, ordered sets of points, cyclic graphs, balanced trees, recombining graphs, meshes, lattices and various hybrids or combinations of such representations. Each node represents one point in the data representation that is implemented using a data structure. The topology of the data representation is determined by the interconnections

among the data structures. A node contains data in various forms, depending on the particular problem to be solved. Choices from among possible data representations are made based upon the attributes of the particular problem to be solved. Data contained in a node can be in the forms of numeric tags, character tags, boolean flags, numeric values, character values, objects IDs, database-record IDs, simple arrays, variable-density multidimensional arrays, symbolic functions, mathematical functions, connection pointers to other nodes, function pointers, lookup-table list pointers, linked-lists, or even pointers to other solution spaces or data representations.

The instantiation engine (IE) provides instructions for the instantiation unit (IU) that creates and deletes nodes (the IU and other machine units are shown in FIG. 2). The population engine (PE) provides instructions for the population unit (PU) that stores data into nodes, and the arithmetic and logic unit (ALU) that emulates a more traditional hardware-implemented ALU. The navigation engine (NE) provides instructions for the navigation unit that reads selected nodes. The evolution engine (EE) provides instructions for updating the contents of the IU and the PU. The configuration engine (CE) provides instructions for the solution-space configuration unit (SCU), which allocates memory for the data nodes and the node index. The SCU also stores configuration parameters for every aspect of the architecture.

The configuration engine (CE) modifies a data representation(s) to create a topology tailored to the problem at hand. When creating this topology, the CE chooses from among a plurality of available topologies and modifies a chosen topology or topologies to suit the given problem. The CE then stores the chosen data representation parameters, and hardware configuration parameters, into the SCU.

The virtual operating system, including its component parts, interacts with the VM via the virtual assembler. The virtual assembler is analogous to a conventional assembler or compiler in that it converts function calls written in a high-level programming language into commands that the machine can understand and process. In this case, the commands are in a format the virtual machine can process.

The NVSI virtual machine (VM) interacts with the platform drivers. The platform drivers allow the virtual machine to interact with the operating system resident on the host computer. The platform drivers interact with one or more underlying hardware platform CPUs via a platform assembler, which converts commands from virtual machine-level function calls to commands that the platform-specific operating system and hardware can understand and process.

The virtual operating system has the ability to create multiple threads to perform tasks concurrently. When a new thread is created, a new virtual central processing unit (VCPU) is created along with the thread. Newly created VCPUs are not complete copies of the entire virtual machine. VCPUs contain only the components necessary for their respective processing tasks, auch as the IU, PU, and NU. Certain components of the VM, such as the index memory, data memory, the configuration unit, and the network control unit (comprising the backbone 'core' of a CPU), are not typically duplicated in threads. The resources and services provided by such components are shared among the other components of the virtual supercomputer.

A functional block diagram of the components and interconnections within the virtual machine (NVSI Virtual Machine, as denoted by the bold-bordered box in FIG. 1, is shown in FIG. 2.

The Solution-space Configuration Unit (SCU) contains the index base register (IBR) stack, the index-memory allocation and data-memory allocation registers (IMAR, DMAR), the node data-structure definition register (DDR), the data definition array (DDA) memory, the field type (FT) table, the number of nodes register (NNR), the data definition count register (DDCR), and the parameter configuration register (PCR). The SCU is where the configuration structures of the virtual machine are stored. The PCR contains information that defines various attributes of the virtual machine by defining the structure of the data words used. This configuration can be altered by users or an application program at startup, or even during runtime execution.

The IBR stack is a set of IBRs that provides an indexed addressing system for memory access. Each virtual register stores a base address that specifies a boundary for a segment of the virtual supercomputer's memory space. Offsets may be calculated by taking this base address value and adding to it the value from the virtual node address (VNA) field of the node index word (IW) stored.

The DDA is a table of data-structure definition words (DDW), each identified by a unique integer data definition number (DDN). When a 'store data definition word' instruction is executed, if the DDN indicates that the DDW is new, the word is written into an available free memory location in the DDA. If the DDN indicates the word is not new, the previous version of the DDW is overwritten with the new version. The DDW to write into the table is located in the DDR.

The FT table in the SCU stores a list of preset data word field types, such as tag, flag, character, integer, fixed-point, floating-point, function pointer, node pointer, and list pointer.

This table defines the valid field types that may be contained in a DDW (and may be extended via settings in the PCR).

The NNR is a virtual register that stores the current count of non-null nodes. It assists in the numbering and creation of new nodes as such nodes are instantiated, and serves effectively as a measure of the size of the solution space.

The DDCR is a virtual register contained within the SCU that contains the total count of active data definitions. This information is used for DDA operations.

The PCR stores the basic parameters that define all configurable (and thus alterable) elements of the virtual supercomputer CPU. Such configurable elements include maximum field lengths, maximum register lengths, the number of registers in a stack, or the sizes of arrays and tables.

The Instantiation Unit (IU) creates the nodes and makes space for the nodes in the data representation. The IU contains one node index word (IW) in the node index word register (IWR). The IW contains a null flag that is set when a delete instruction is executed for a specified node. The next field contains the DDN. Following the DDN is a field that specifies the length of the data word. Next is the VNA, followed finally by an application-defined field (ADF). The ADF can be used for special purposes defined by the specific application.

The node-counter (NC) register is a virtual register containing a pointer to the next node. The NC contains a node number that is a non-negative integer specifying the relative address of the corresponding IW in the node index memory (NIM).

The Population Unit (PU) contains a node data word (DW) stored in a virtual register labeled the node data-word register (DWR-P). A DW may be fixed length or variable length.

A fixed length flag indicates the type of a particular DW. The DW stored in the PU is used when populating the solution space (Node Data Memory, NDM) with nodes. The PU evaluates the data structure for a given node. The results of such evaluation are stored into a data word whose location in the NDM is pointed to by the index word. Every data word has a unique address in the NDM that is the VNA.

The navigation unit (NU), like the PU, contains a node data word (DW) stored in a virtual register labeled the node data word register (DWR-N). This value is used when navigating through the solution space.

The node index memory (NIM) contains the node free index (NFI) and the next free node registers (NFNR1 & 2). The NFI stores the current intervals of free nodes in the node index space. The NFNRs are loaded from the NFI, and store the beginning and the end of a range of free nodes. The primary use of the NFNRs is during instantiation operations where unused node index words are overwritten.

The arithmetic and logic unit (ALU) is a software implementation of some functions that are often implemented in hardware. It contains an adder/multiplier, a logic evaluator, an arithmetic register (AR) stack, a lookup table index, a function index, and an ALU memory. It allows as well for 'pass-though' of arithmetic operations to the underlying hardware CPU.

The physical memory controller (PMC) operates between the NIM and the NDM. The PMC controls the use of physical memory devices such as random access memory (RAM), disk drives, optical storage drives, and other physical memory devices which may be available to store data.

The network control unit (NCU) handles the low-level details of sending out data and processes to be processed. It in turn is controlled by a network manager. These two units

handle the tasks of separating tasks to be run concurrently, load balancing, and other network and concurrency-related management tasks.

The CE store configuration parameters in the PCR, and also creates and stores data definition words (DDWs) in a manner depicted by the pseudocode in FIG. 3. The engine begins by entering a loop. This loop executes once for each of a specified number of data-word architectures in the domain solution space modeled within the data representation. Within each iteration of the loop, the CE creates a DDW in register DDR according to the parameters specified by the domain application program. The CE next stores the DDR into the DDA in the configuration unit. The CE then continues its processing by executing the next iteration of the loop. The CE finishes its execution when it has executed the loop the specified number of times.

The IE creates nodes in a manner depicted by the pseudocode in FIG. 4. The engine begins by entering a loop. This loop executes once for each of a specified number of nodes to be created in the domain solution space modeled within the data representation. Within each iteration of the loop, the IE creates an IW in register IWR in the IU. The IE next stores the IWR into index memory at a node number indicated by the node counter. The IE then allocates space in data memory at a virtual node address (VNA) calculated by the IM internal memory manager based upon parameters in the corresponding DDW word. The IE then continues its processing by executing the next iteration of the loop. The IE finishes its execution when it has executed the loop the specified number of times.

The population engine (PE) evaluates and stores nodes in a manner depicted by the pseudocode in FIG. 5. The PE begins by entering a loop. This loop executes once for each of a number of nodes. The PE reads an IW from index memory (NIM) at the specified node

address. The PE next reads the DDW pointed to by the DDN in the IW. The PE then evaluates all fields in the data word according to the corresponding DDW. The PE then creates a data word in the data word register (DWR-P) in the population unit. If the length of the data word has changed, then the internal memory manager computes a new VNA, stores the new VNA into the corresponding IW and updates the VNA in the IWR, and stores the DWR-P into data memory (NDM) at the new VNA. If the length of the data word has not changed, the PE stores the DWR-P into data memory at the old VNA.

The navigation engine (NE) finds and reads a node data word in a manner depicted by the pseudocode in FIG. 6. The NE gets the selected node number from the domain application program. The NE then reads the IW from index memory at the position specified by the node counter. The NE reads the data word at the corresponding VNA into the DWR-N.

The evolution engine (EE) adds, deletes, or modifies nodes in a manner depicted by the pseudocode in FIG. 7. The EE begins execution by getting a selected node number from the domain application program. The EE then gets the evolution condition from the domain application program. The evolution condition specifies whether the EE is to add a new node, delete an existing node, or modify an existing node. If the condition specifies that the EE is to add a new node, the EE calls the instantiation procedure for the specified node number. The EE then calls the population procedure for the same node according to parameters specified by the domain application program. If the condition specifies that the EE is to delete an existing node, the EE calls the instantiation procedure in delete mode for the specified node, and updates the NFI. If the condition specifies that the EE is to modify an existing node, the EE calls the navigation procedure for the specified node number. The EE

next modifies fields in the DWR-P as specified by the domain application program. The EE then calls the population procedure for the specified node number. When the activities required by the given condition are completed, the EE completes its execution.

Further details regarding this invention are contained in appendices A and B, which form part of this patent application.